Service Line Engineering in Practice: Developing an Integrated Document Processing SaaS Application

Stefan Walraven
Dimitri Van Landuyt
Fatih Gey
Wouter Joosen

Report CW 652, November 2013, revised February 2014

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Service Line Engineering in Practice: Developing an Integrated Document Processing SaaS Application

Stefan Walraven
Dimitri Van Landuyt
Fatih Gey
Wouter Joosen

Report CW 652, November 2013, revised February 2014

Department of Computer Science, KU Leuven

Abstract

This report presents the application of our service line engineering method for developing and managing customizable, multi-tenant Software-as-a-Service (SaaS) applications. This approach of service line engineering is feature-oriented and highly integrated, in the sense that the feature-level variability that is introduced in the early development stages is consistently and explicitly supported in each of the subsequent stages, also in the run-time environment. The method is generic in the sense that each stage is open for existing work in the state of the art to be leveraged upon, yet it imposes some specific constraints and some enablers, which are crucial to obtain the desired variability of a service line.

The example in this report shows a practical application of the service line engineering method in the domain of online document processing. Concretely, it presents and discusses the six activities of the method, including the relevant artifacts and a discussion on the key design decisions. In addition, we present some auxiliary solutions in order to make the generic method effective in practice.

Keywords : Multi-tenancy, Software as a Service, Service Line Engineering.

Service Line Engineering in Practice: Developing an Integrated Document Processing SaaS Application

Stefan Walraven, Dimitri Van Landuyt, Fatih Gey, Wouter Joosen

November 2013, revised February 2014

Abstract

This report presents the application of our service line engineering method for developing and managing customizable, multi-tenant Software-as-a-Service (SaaS) applications. This approach of service line engineering is feature-oriented and highly integrated, in the sense that the feature-level variability that is introduced in the early development stages is consistently and explicitly supported in each of the subsequent stages, also in the run-time environment. The method is generic in the sense that each stage is open for existing work in the state of the art to be leveraged upon, yet it imposes some specific constraints and some enablers, which are crucial to obtain the desired variability of a service line.

The example in this report shows a practical application of the service line engineering method in the domain of online document processing. Concretely, it presents and discusses the six activities of the method, including the relevant artifacts and a discussion on the key design decisions. In addition, we present some auxiliary solutions in order to make the generic method effective in practice.

Contents

1 Introduction

This report describes the practical application of the service line engineering (SLE) [26] method for developing and managing customizable, multi-tenant Software-as-a-Service (SaaS) applications. It serves to document how we applied this method to develop a prototype for a concrete SaaS application and provides all details of this. More specifically we apply it on a multi-tenant SaaS application for B2B document processing, which is based on the offering of an industrial partner in the iMinds-CUSTOMSS project [4].

Section 1.1 summarizes the service line engineering method, and Section 1.2 presents the example SaaS application. The remainder of this report is structured as follows. In Part I, we present some essential, yet auxiliary solutions that were necessary to make the generic SLE method effective in practice. Part II describes the practical application of the SLE method for developing the prototype of the document processing application, and briefly discusses how the SLE method supports evolution and maintenance. Finally, Section 11 concludes this report.

1.1 Summary of the Service Line Engineering Method

The service line engineering (SLE) [26] method[1] aims to facilitate the development and management of customizable, multi-tenant SaaS applications, without compromising the essential benefits of scale associated to cloud computing. It combines the benefits of software product line engineering (SPLE) [3, 21] with those of multi-tenancy [1, 13] to enable efficient customization of SaaS applications tailored to the tenant-specific requirements. We define a *service line* as a SaaS application that is built as a software product line consisting of customizable services that can be dynamically composed and configured based on the tenant-specific requirements, with the major difference that one single instance is supporting the different application variants.

Concretely, the SLE method is feature-oriented and highly integrated: the feature-level variability that is consistently and explicitly supported in each of the development and deployment stages, even at run time. This is a key difference w.r.t. traditional SPLE. Instead of delivering a dedicated application product for each tenant (cf. the application engineering phase in SPLE), the entire service line (including all variations) is instantiated and deployed only once and simultaneously shared by all tenants. Specific software variants are activated at run time within the same SaaS application instance.

The SLE method is depicted in Fig. 1, presenting the individual development and management activities and their input and output artifacts of relevance to the method. It consists of four high-level processes: (i) a feature-driven service line development process, (ii) the deployment of the developed service line, (iii) service line configuration for each tenant, driven by their respective requirements, and (iv) run-time composition of the appropriate software variants into the deployed service line based on the tenant-specific configurations. The initial investment is in Service Line Development and Deployment, while Service Line Configuration and Composition become relatively straightforward by leveraging on the investment of the former two. We briefly introduce the different activities

[1] http://distrinet.cs.kuleuven.be/projects/CUSTOMSS

of each high-level process.

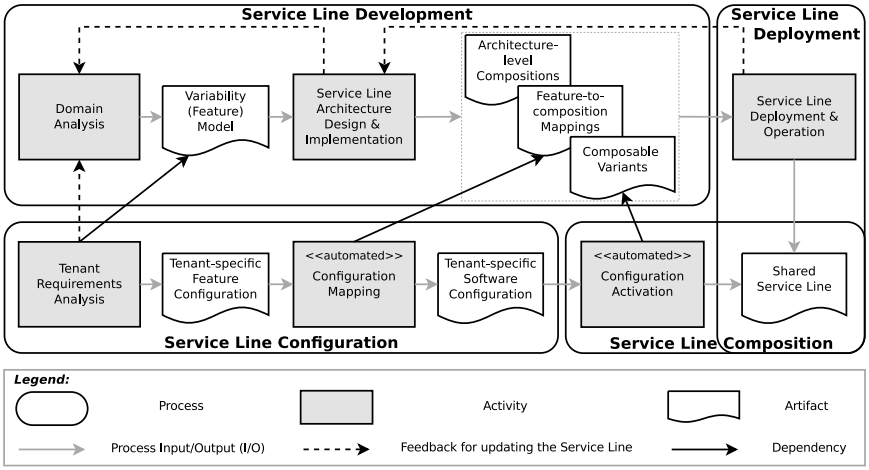

Figure 1: Overview of the Service Line Engineering method.

Service Line Development The process of service line development involves the initial development of the entire service line as well as its further evolution and maintenance. The SaaS architect and SaaS developers are mainly responsible for the different activities in this process.

Engineering a service line starts with the activity of `Domain Analysis` to obtain the essential characteristics of applications in a particular domain (e.g. document processing), and is quite similar to SPLE. Variability analysis is part of domain analysis and has (in our method) a feature-based variability model as end result. The SLE method requires support for versioning of the feature model itself and the explicit representation of non-fuctional requirements (i.e. SLAs).

In the next activity, `Service Line Architecture Design & Implementation`, the initial architecture of the multi-tenant SaaS application is created and implemented. The service line architecture makes variability explicit by means of variability-supporting views. More specifically, apart from the typical architectural views, it also models the variation points and all variants, independent from the underpinning (dynamic) composition technologies. Additionally, feature-to-software-composition mappings are defined between features from the feature model and components in the architecture. Finally, the different software variants are implemented in a modular way to improve composeability and reusability.

Service Line Deployment During the `Service Line Deployment & Operation` activity, the SaaS operator creates an instance of the service line, which is deployed and offered as a multi-tenant service on top of a powerful platform. After deployment, the focus shifts towards operation, i.e. applying updates and upgrades (evolution and maintenance) while avoiding service disruption. The deployment and operation activities highly rely on the presence of a suitable

SaaS middleware (e.g. to support multi-tenancy, dynamic composition and versioning).

Service Line Configuration Instead of delivering a dedicated application product for each tenant, tenant-specific configurations are created that are immediately effective and co-exist in the running service line. The `Tenant Requirements Analysis` activity consists of acquiring the tenant preferences and requirements by means of a feature-oriented configuration interface (based on the feature model as defined during domain analysis). The outcome is a feature configuration containing a set of features and potentially some feature-specific attributes.

Next, each tenant-specific feature configuration is automatically mapped to a software configuration (cf. `Configuration Mapping`) based on the mappings that have been defined during service line development. A software configuration specifies a composition of particular software variants, which correspond to the selected set of features.

Service Line Composition Finally, during `Configuration Activation`, software variants are dynamically activated and composed into the service line to match the scope of a particular tenant. This is decided on a per-request basis, depending on the configuration of the tenant associated with the current request.

Iterative service line refinement As with many software engineering approaches that deal with evolving requirements [8, 19], it is an essential property of the presented service line engineering method that it is iterative. This is represented by the multiple feedback loops in Fig. 1. For example, the analysis of tenants' requirements might affect the domain analysis and possibly lead to a re-iteration over the development and deployment activities. The experiences gained from executing and monitoring the service line in the deployment activity can give feedback on the implementation of the service line. In addition, upgrades of the platform or libraries can also have impact on the implementation. Similarly, implementation changes such as introducing new dynamic composition mechanisms or performance improvements, can affect the service line architecture and the mappings defined therein. Finally, updating and extending the service line architecture itself might lead to changes in the domain analysis.

1.2 The Document Processing SaaS Application

UnifiedPost[2] is a European SaaS provider that offers B2B document processing facilities to a wide range of companies in very different application domains. This multi-tenant SaaS application supports the creation and generation, the business-specific processing and the storage of millions of business documents per day, such as invoices and payslips, even up to printing and distributing. This is a fairly large-scale and complex SaaS application, which offers roughly 25 high-level, distinct and parameterizable variations (such as document generation, signing and archival), and which services around 150 different tenant

[2] http://www.unifiedpost.com/

organizations. These tenants are able to create, store, manage and send out personalized documents, tailored to their specific preferences. In addition, tenants and their customers can view and manage documents via a web-based interface, and download them from the document store.

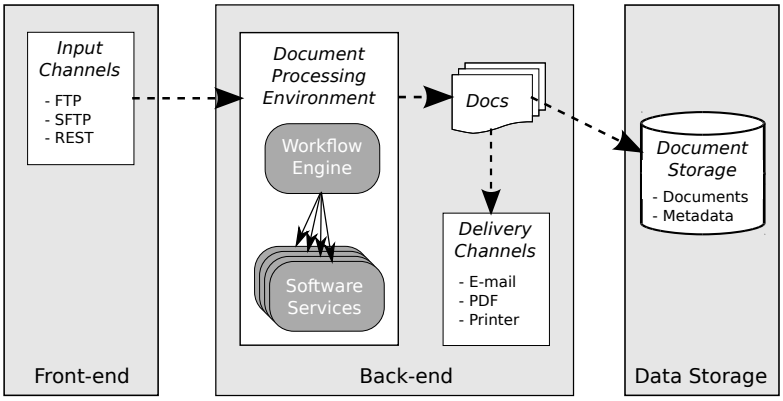

Figure 2: High-level overview of the online document processing application.

An overview of this multi-tenant SaaS application is presented in Fig. 2 and consists of four main subsystems: a set of input channels, a document processing environment, a set of delivery channels, and document storage. The tenants use the input channels to deliver raw data, potentially using a multitude of formats. These input data sets contain information such as customer names, billed items, and other customer- and organization-specific data. The core part of the SaaS application is the document processing environment with a workflow engine that executes workflows to validate and verify input, generate document layouts, and to process and store business documents. All these actions are performed by different software services, e.g. for generating documents and archival. The final documents are delivered to the customers and end users via an output channel, for example email or printed mail. The document storage is responsible for storing the input and output data, and all associated metadata.

Part I
Auxiliary Solutions to Support SLE

While validating the feasibility of the end-to-end integrated SLE method, we had to bridge some gaps in the current state of the art by complementing our generic method with some essential, yet auxiliary elements: (i) we have a basic feature meta-model that explicitly represents SLAs and that supports versioning (Section 2), (ii) we have created a variability meta-model that supports co-existing configurations and that is open for different composition technologies at different levels of granularity (Section 3), (iii) we have specified a generic software configuration representation, independent from underpinning technologies and platforms, to enable efficient management and reuse of configurations and software variations (Section 4), and (iv) we have developed a basic set of middleware services to facilitate the development and management of service lines, and to minimize the time and cost to provision tenant-specific application variants via an automated configuration process (Section 5). In this part of the report we describe these auxiliary solutions in more detail.

2 Feature Meta-model

In our SLE method, the domain analysis activity focuses on creating a *variability model* to represent the domain-specific commonalities and variabilities in a service line. This activity does not differ substantially from domain analysis in traditional SPLE approaches, except for two essential differences that can be attributed to the multi-tenant SaaS context:

- The context of multi-tenancy introduces additional variability, driven by the *non-functional requirements* of the different tenants (e.g. availability and performance). Our SLE method requires that this additional variability is explicitly represented in the variability model.
- To support evolution of SaaS applications in the form of continuous updates ("develop once, adapt/evolve forever"), while minimizing downtime, *versioning* is an essential enabler. Therefore, versioning support should be present in every stage of the method, and thus also in the variability model.

In this report, we applied a feature-based approach [14, 15] to model variability, and thus the relevant output consists of feature models. To support the service line variability required for our method, we used the feature meta-model presented in Fig. 3. A `Feature` declaration consists of an identifer or unique name, a version number and an optional description. `Relations` between different features can be categorized in parent-child relations, such as mandatory, optional and alternative, and constraints, such as dependencies ("requires") and conflicts ("excludes"). In addition, the SaaS provider can define additional attributes to enable parameterization of the features. It is a rather elementary feature meta-model, but it suffices for the document processing application and

it does support non-functional requirements (SLAs) and versioning by means of the `version` property[3], as required by our method.

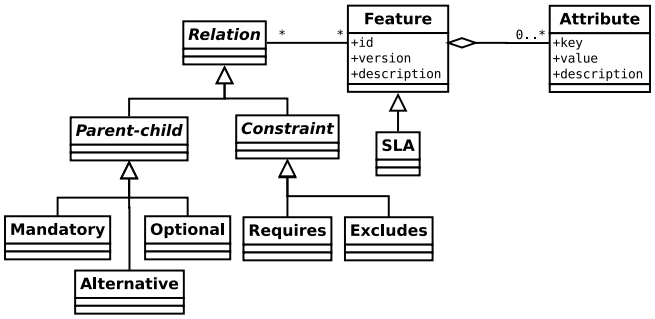

Figure 3: Meta-model for feature models.

During the activity of tenant requirements analysis, a configuration interface is offered to the tenant administrator to express the tenant's preferences and requirements. This configuration interface builds further on the feature-based approach that we applied during the domain analysis by creating a feature model for the service line. By selecting the appropriate features from this feature model and configuring specific feature attributes, a *tenant-specific feature configuration* is created for each tenant.

Fig. 4 presents the feature configuration meta-model that we used for this report, which is an extension to the feature meta-model of Fig. 3. It describes the different concepts and their relations with respect to the tenant-specific configuration of service lines in terms of features. For each `Service Line`, a `Tenant` can specify a `Feature Configuration`. A `Feature Configuration` is derived from a feature model. It contains the set of features selected by the tenant administrator and, when applicable, specifies the value for a feature attribute to parameterize it. Similar to features, feature configurations are also versioned. This allows tenants to revert to earlier configurations when desired.

3 Variability Meta-model

With respect to the activity of service line architecture design, the SLE method requires the service line variability in separate architectural views, enabling the SaaS architect to separately manage and design different variations and versions in the service line. Furthermore, the composability of features and their corresponding software variants should be ensured, i.e. feature implementations can be separately developed and later be composed into a working system.

To realize this in our prototype, we used the generic variability meta-model as presented in Fig. 5 to provide the SaaS architect with the means to build an overview of the architecture of a multi-tenant SaaS application and its variability. A `Service Line` consists of a set of customizable compositions (`Composition`), without a fixed level of granularity. More specifically, it supports a hierarchical structure, represented by the composite structure around

[3]Detailed versioning support (e.g. diff and merge support of feature models) would be useful in the context of service line evolution, but is out of scope for this report.

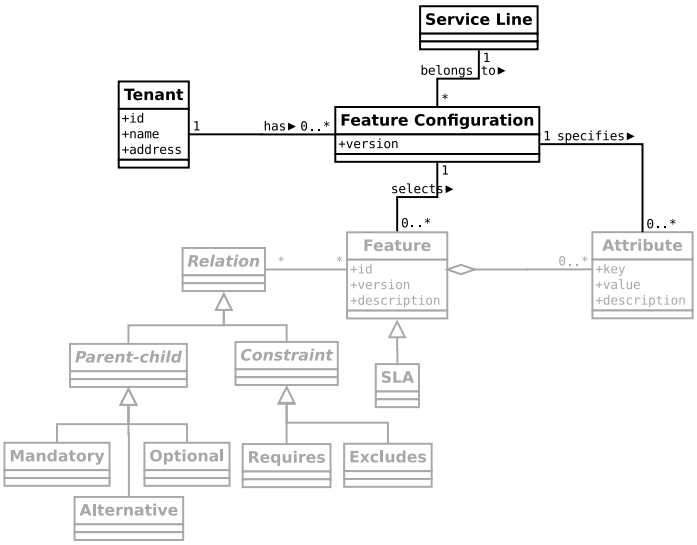

Figure 4: Meta-model for tenant-specific feature configurations (extension to Fig. 3).

`Component` and `Composition`. This allows variations to be introduced at different levels of abstraction: from the top level of coarse-grained system-level components, down to the level of fine-grained compositions. Each level can use different implementation and customization techniques [5], for example at the level of services in a workflow or components in a service. The points of customization are represented by means of `Variation Point`s. For each `Variation Point` at least one `Variant` exists that can be used in the composition. A `Variant` consists of one or more `Components`. A component has a unique ID, a set of properties, and a reference to a software artifact.

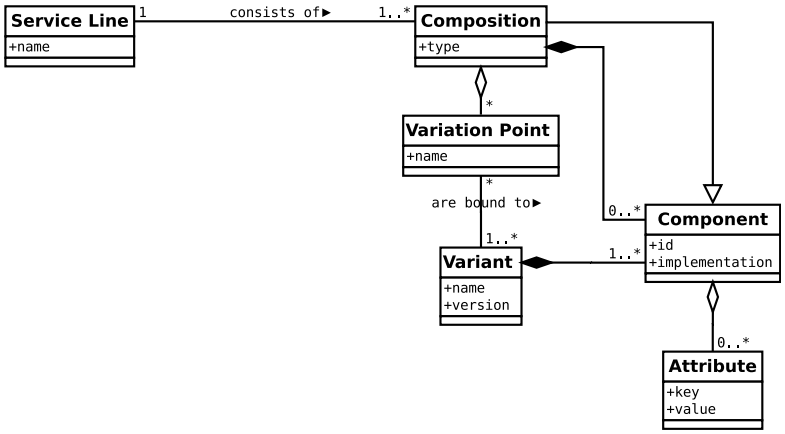

Figure 5: Metamodel for variability in service line architectures.

Furthermore, the meta-model makes abstraction of the specific composition mechanisms and technologies that will be used to associate variation points with specific variants, for example aspect-oriented programming (AOP) [17], dependency injection [9], and component-based software development (CBSD) [22]. The type of composition technology is indicated via the `type` attribute of `Composition`.

During the automatic activity of configuration mapping, tenant-specific feature configurations (cf. Section 2) are transformed into a software configuration. These tenant-specific software configurations co-exist in the running service line. We extended the variability metamodel with the concept of a software configuration (see Fig. 6). A `Software Configuration` defines for a particular tenant which specific variants should be (dynamically) composed into the service line. The core building blocks of a software configuration are `Binding`s, which define which specific variants should be bound to the different variation points. Bindings can also specify the values of attributes belonging to components.

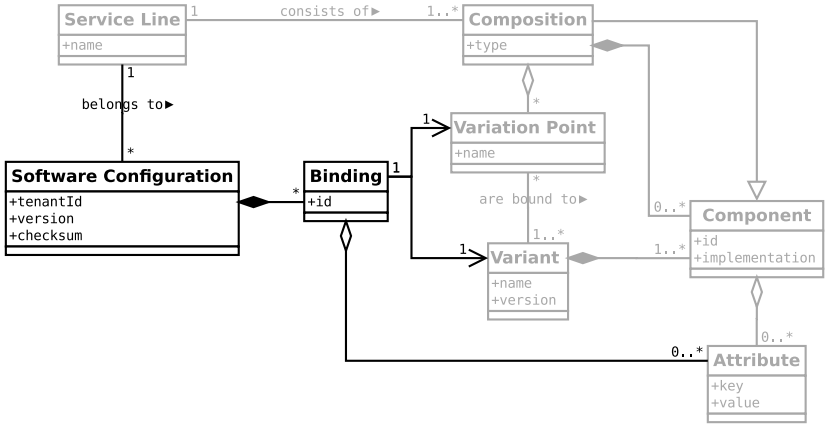

Figure 6: Metamodel for tenant-specific software configurations (extension to Fig. 5).

4 Feature Mapping Specification

In order to enable the automated transformation of feature configurations to software configurations (cf. activity of configuration mapping), machine-interpretable mappings between features and software variants have to be defined by the SaaS architect for each feature in the feature model. These *feature-to-software-composition mappings* thus realize specific traceability links between features (as defined in the meta-model of Fig. 3) and architecture-level compositions (as defined in the meta-model of Fig. 5) by binding a variation point to a specific variant.

To achieve this automated transformation in our prototype, we have defined a custom grammar for these feature-to-software-composition mappings (see Listing 1). A feature-to-software-composition mapping maps a feature identifier `<feature>` (which is a combination of the feature name and its version number

as shown on line 5) to one or more compositions (`<composition>+`), i.e. to those software compositions that realize the specific feature (line 3). Each feature-to-software-composition mapping is given a unique name (`<mapping-id>`) (line 2). To enable reuse of existing feature mappings, mappings defined elsewhere can be imported, as exemplified by the import statement on line 6. Further, a mapping defines the set of compositions to which it applies (lines 7–18). These compositions correspond to the compositions defined in the service line architecture (that complies to the meta-model of Fig. 5) and the mapping specifies which variation points are to be filled in. To fulfill the variation points, the composition element of a mapping (lines 7–12) specifies bindings (`<binding>`). A binding specifies the link between a variation point and a specific variant (lines 13–18). In addition, a composition can also refer to other compositions defined in the mapping by means of their IDs (line 10). This is required for sub-compositions, for example when a workflow (i.e. the root composition) consists of a set of customizable services (i.e. the sub-compositions). Finally, the SaaS provider can specify attributes for compositions as well as bindings, optionally providing a default value.

Listing 1: Grammar for feature mapping specifications (BNF).

```
1  <feature-to-software-composition-mapping> ::=
2   featuremapping <mapping-id> {
3      <feature>, (<mapping>)*, (<composition>)+
4   }
5  <feature> ::= feature <feature-id>-v<feature-version>;
6  <mapping> ::= import <mapping-id>;
7  <composition> ::=
8   composition <id> {
9      (<binding>)+
10     (composition: <id>;)*
11     (<attribute>)*
12  }
13 <binding> ::=
14  binding [<id>] {
15     variationPoint: <string>;
16     variant: <string>-v<variant-version>;
17     (<attribute>)*
18  }
19 <attribute> ::= attribute: <key> [- <value>];
```

Our mapping specification focuses on functional features that can be mapped to a set of architectural components (and their corresponding software artifacts). When this is not the case (e.g. for an availability or performance SLA), these features (and their attributes) should be used as input for the underpinning middleware or a broker (e.g. the monitoring framework or the policy enforcement engine).

5 Service Line Management Support Layer

After the development of the service line, the SaaS operator creates an instance of the service line and deploys it on top of the cloud infrastructure of the SaaS provider (cf. the activity of service line deployment). The selection of an appropriate environment to host the service line has a major impact on the design, the

implementation, as well as the deployment and operation process. For example, the SaaS provider can decide to deploy the service line on existing PaaS or IaaS offerings, or to set up his own cloud infrastructure. However, to develop and host a service line, the necessary middleware support should be available.

The SLE method requires at least the following elements of the underpinning SaaS middleware:

- Versioning support for features and their implementations is required to enable co-existing versions of service line artifacts, and to support traceability.
- Application-level multi-tenancy is a core characteristic of a service line and thus should be supported. As indicated by [25], some existing PaaS platforms already offer built-in support for tenant data isolation, e.g. Google App Engine [12].
- Support for tenant-aware dynamic composition is critical to be able to activate the appropriate software variants at run time, based on the co-existing tenant-specific configurations.

In addition to these enabling middleware services that address the core requirements for service lines, we have developed a generic support layer to facilitate the development and management of service lines (see Fig. 7). This layer offers several service interfaces to the different stakeholders and to the application layer:

- The `IFeatureManagement` interface provides SaaS providers –and more specifically, the *SaaS architect/developer*– with a service to manage the feature model and the feature mapping specifications. The SaaS provider has to specify a feature-to-software-composition mapping for each feature and upload it. Next, the feature mappings are parsed, verified, persisted and activated.
- The `ITenantManagement` interface is accessed by the *tenant administrators*, which are (in the context of service lines) employees of the tenant organization (cf. self-service). This interface enables the registration of new tenants to the service line. For example, a new tenant has to provide his name and billing address, select a unique ID and a domain name. This domain name will be used by the end users of the tenant organization to access the shared service line application. In addition, tenant administrators can use this interface to customize the SaaS application by selecting features based on the tenant's preferences and by parameterizing these features. After this activity, the resulting feature configuration is verified for correctness by consulting the interdependencies between the features (as specified in the feature model), enabling immediate (online) feedback to and response by the tenant administrator.

  Next, the feature configuration is automatically transformed into a tenant-specific software configuration based on the applicable feature mapping specifications (by the internal `ConfigurationMapping` service). When both the configurations are valid, they are persisted and activated into the running SaaS application.

- The `ITenantConfigurationRetrieval` interface is offered to the multi-tenant SaaS application, more specifically to allow the look-up of tenant-specific software configurations and to enable the run-time composition of the SaaS application.

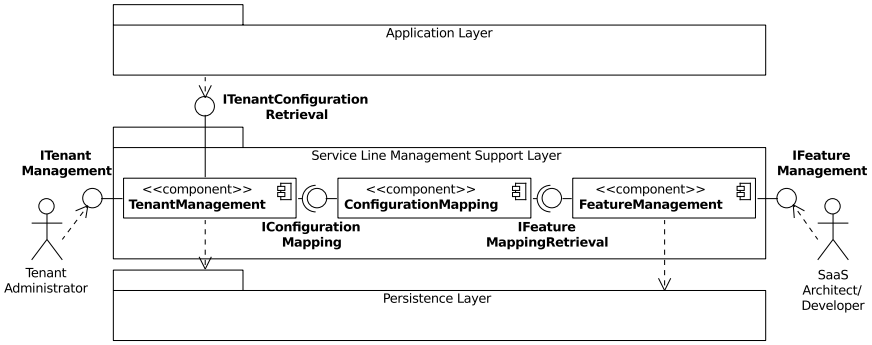

Figure 7: Enabling services for service line management.

We have implemented these services as a reusable middleware layer using Java EE 6. The front ends are realized with Java Servlets, JSPs, and RESTful services; the business logic is implemented using session beans; for persistence we used entities that are stored into a MySQL database (accessed through JPA [6]). Furthermore, we have developed the custom grammar to express the feature-to-software-composition mappings (see Section 4) using ANTLR [20]. This middleware layer is designed for reuse across different service lines.

Part II
Application of SLE on the Document Processing SaaS Application

The second part of this report describes the practical application of the SLE method in the domain of online document processing. More specifically, we focus on the scenarios of developing a service line in Section 6, provisioning new tenants in Section 7, and updating tenant-specific configurations (i.e. addressing changing requirements) in Section 8. For each of these scenarios, we discuss the relevant activities and output artifacts.

Furthermore, we briefly discuss how the SLE method supports evolution and maintenance of a service line. Concretely, we focus on the scenarios of supporting new requirements of tenants in Section 9, and updating and maintaining the SaaS application (i.e. upgrading to new versions of the platform, libraries etc.) in Section 10.

6 Initial Development of Service Line

This section describes the initial design and development of the document processing service line. This relates to the activities of domain analysis, service line architecture design & implementation, and service line deployment & operation.

6.1 Activity 1: Domain Analysis

As a first activity, we gained knowledge on the document processing domain via on-site interviews with UnifiedPost. Based on the results of these interviews, we distilled the functional and non-functional characteristics of their online document processing application and we determined the significant variation points. The output of this activity consisted of a wide variety of artifacts, such as domain models and use cases. However, with respect to the service line engineering method, the key output of this activity is a feature-based variability model that represents the domain-specific commonalities and variabilities of the document processing SaaS application.

Requirements. Concretely for the prototype, we focused on the requirements of one of the tenant clusters of UnifiedPost. We present two examples of tenant companies in this cluster, which have different requirements regarding document processing. These requirements are based on actual tenants and thus are sufficiently representative examples of realistic variations. For non-disclosure reasons, we anonymized the names of the tenant companies.

Tenant A is a temporary employment agency that uses UnifiedPost's SaaS application to process the payslips of all its employees. On a regular basis, Tenant A submits a set of payslip documents (PDF), along with some meta-data. These payslips need to be printed and distributed among the different employees based on the associated meta-data. Because Tenant A has a large

amount of employees and it requires that all payslips are delivered within a strict period of time, document processing should occur with a guaranteed throughput. Evidently, Tenant A is prepared to pay premium fees to obtain such a specific SLA in terms of throughput and deadlines.

Tenant B is active on the financial services market and uses UnifiedPost's SaaS application to process its invoices and to distribute them to corporate customers. In contrast to Tenant A, Tenant B only provides raw data in XML format (i.e. document data and meta-data) as input, and thus requires the generation of the invoices. The generated documents should be delivered to the customers (i.e. end users) of Tenant B, depending on their preference: via email or on paper (printed mail). In addition, the generated invoices should be signed and archived securely for a period of 24 months, and delivered as fast as possible. However, Tenant B does not want to pay an extra charge for a guaranteed throughput.

Feature model. To create the feature model for our prototype, we used the feature meta-model presented in Section 2. Figure 8 presents the feature diagram that covers a subset of the variability in the document processing SaaS application. It matches the requirements of the cluster of tenants that we investigated (see above). We used FeatureIDE [16] as modeling tool to create this feature diagram, which does not support all model elements from our feature meta-model (e.g. attributes and version numbers). Therefore, these elements are not represented graphically in Fig. 8, but they do exist in the underlying model specification. More expressive approaches for feature modeling, e.g. TVL by [2], do offer support for attributes.

At the top level, the service line offers features related to `Input`, `Processing`, `Distribution` and `Archival`. Regarding `Input`, the tenant should select (i) an input protocol (i.e. how the raw data will be uploaded), (ii) the format(s) of the input data, and (iii) how a batch of input data is demarcated. Batch demarcation occurs by means of flag files per input file or a single flag file for the whole batch.

`Processing` groups the following sub-features: (i) different alternatives to acquire meta-data from the raw input data (under `Meta_Acquirement`), (ii) an optional feature for document generation based on different templates (under `Document_Generation`), (iii) an optional feature named `Signing` for signing documents using the SaaS provider's or tenant's certificate, and (iv) performance SLAs (`Throughput`) that offer different performance levels. The `Signing`, and `Custom` (document generation) sub-features require an attribute to be provided, respectively the certificate name and the template for document generation.

In terms of `Distribution`, the tenant can select from several delivery channels: email (with document in attachment), printed mail, or Zoomit (i.e. online banking service that securely transfers documents to an inbox linked to the receiver's bank account).

Finally, the optional `Archival` feature enables tenants to archive documents conform legislation. This feature has as required attribute the duration of the archival period.

The logical expression at the bottom of the figure ($\neg Archival \vee Signing$) represents a dependency between features: documents can only be archived if they are signed. So, selecting the `Archival` feature implies that the `Signing`

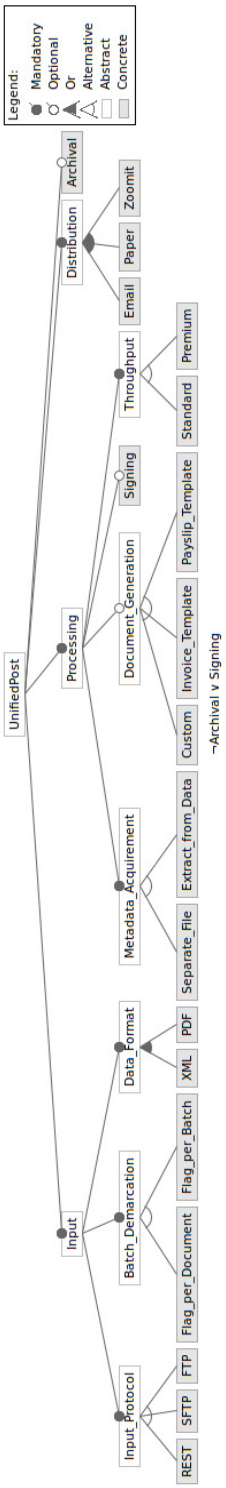

Figure 8: Feature diagram of the document processing SaaS application.

sub-feature must be selected as well.

6.2 Activity 2: Service Line Architecture Design & Implementation

The domain analysis activity is followed by the design and implementation of the document processing service line. The design decisions made in UnifiedPost's current SaaS application influenced the design of our prototype. Obviously, we created several architectural views [18] during the design activity, for example using component diagrams, sequence diagrams etc. In this report, however, we focus on representing and realizing variability. The relevant artifacts are the architecture-level compositions, the feature-to-software-composition mappings, and the composable variants. A deployment diagram is provided in Section 6.3, as part of the deployment activity.

Architectural Design. Fig. 9 shows a high-level overview of the document processing application, which is designed as a customizable workflow. This workflow is triggered when jobs containing raw data (corresponding to multiple documents), are uploaded by the tenant. These jobs are submitted at a pre-processing component and passed on to the workflow engine. In the first step of the workflow, metadata is acquired from the raw input data. Subsequently, the output documents are (optionally) generated based on this metadata and according to a built-in or custom template. Then, the generated documents are delivered to the appropriate recipients, using one of the available delivery mechanisms. Notice that multiple jobs of different tenants are processed concurrently by this multi-tenant SaaS application.

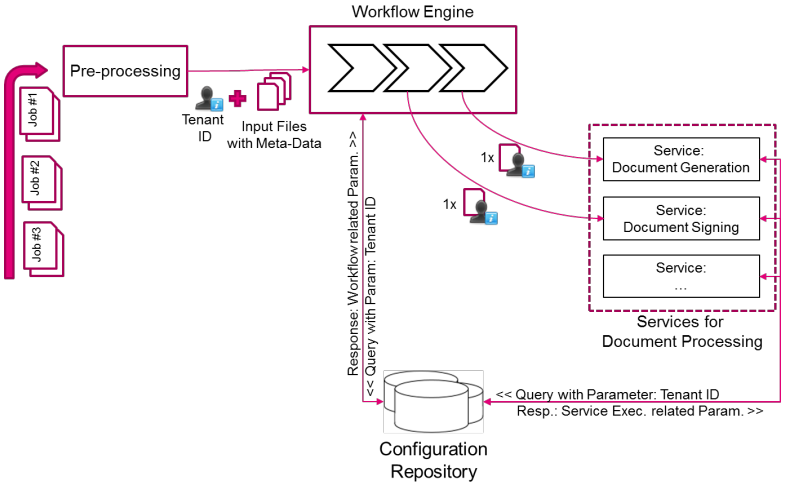

Figure 9: High-level overview of the online document processing application [11].

To represent variability in the architecture of the prototype, we used the variability meta-model that we proposed in Section 3. In general, the document processing application requires support for variability at two different levels of

abstraction. The customizable workflow is modeled as a composition of services. The actual composition occurs at run time based on the tenant-specific requirements that are applicable for the current request. Therefore, the workflow has several variation points. Such a variation point is a service type and each variant is a particular service implementation. The composition type is thus a regular service composition or orchestration. For example, the different 'distribution' sub-features result in a service composition with different variants for the 'delivery channel' service type.

Further, each service implementation itself is also modeled as a composition, which again can contain variation points. In this case, we apply a composition of components via dependency injection [9]. For example, the document generation service offers several strategies for document generation, which require not only different layout templates but sometimes also different supporting technologies (e.g. support for XSL-FO).

Concretely, our service line prototype consists of two compositions: (i) the `Preprocessing` service that accepts the incoming jobs, and (ii) the actual `DocumentProcessing` service that represents the document processing workflow and that is called indirectly by the `Preprocessing` service. The former composition has only one level of variability, namely at the service level.

Feature-to-software-composition mappings. We have specified a mapping of each feature (i.e. those defined in Fig. 8) to components and compositions defined in the service line architecture. For example, Listing 2 shows a mapping that defines which variants to use in case the `Separate_File` feature (line 2) or the `Extract_from_Data` feature (line 20) has been selected by the tenant administrator. The workflow for document processing is defined as the `DocProcessingWorkflow` composition, with `Metadata` as its relevant variation point (lines 5 and 23) and `MetadataAcquirementService` as the corresponding variant for these two features (lines 6 and 24). This variant, however, consists of another (customizable) composition, called `MetadataAcquirement`. Therefore a dependency is added to this composition via its ID (lines 8 and 26). The latter composition has a variation point for the algorithm to acquire the meta-data from the input data. The algorithm that extracts the meta-data from a separate file, requires an additional attribute to define the name of the file containing the meta-data (line 14).

For the sake of readibility, we placed the remaining mappings in Appendix A. Listings 4, 5, and 6 show the feature-to-software-composition mappings for features related to the preprocessing service, while Listings 2, 7, 8, 9, 11, and 10 provide the mappings for the features related to the document processing workflow.

Implementation. During the final step of this activity we have developed a prototype of the document processing SaaS application in Java, implementing the overall architecture comprising the different software variants (corresponding to the features identified during the domain analysis).

In the implementation, the main business logic of the document processing case is modelled as two workflows using the workflow modeling and execution engine jBPM (see Fig. 10). The outer workflow first preprocesses the input data of uploaded jobs and then invokes the inner workflow for each individual docu-

Listing 2: Feature mappings for the Metadata Acquirement sub-features.

```
featuremapping MetadataAcq_File {
    feature Separate_File-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: Metadata;
            variant: MetadataAcquirementService-v1;
        }
        composition MetadataAcquirement;
    }
    composition MetadataAcquirement {
        binding {
            variationPoint: AcquirementStrategy;
            variant: MetadataFromFile-v1;
            attribute: fileName;
        }
    }
}

featuremapping MetadataAcq_Extract {
    feature Extract_from_Data-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: Metadata;
            variant: MetadataAcquirementService-v1;
        }
        composition MetadataAcquirement;
    }
    composition MetadataAcquirement {
        binding {
            variationPoint: AcquirementStrategy;
            variant: ExtractFromData-v1;
        }
    }
}
```

ment. The actual document processing, e.g. meta-data acquirement, document generation and distribution, is performed by the inner workflow. More details on the implementation of this customizable workflow can be found in [11]. The different services that are used in this workflow are implemented as RESTful web services (stateless). For the run-time composition within the services, we relied on dependency injection and reflection.

6.3 Activity 3: Service Line Deployment & Operation

After the implementation of the service line, an instance of this service line can be created and deployed, so that becomes accessible by all tenants. This activity also covers the aspect of monitoring the running instance, which software configurations are active, etc., as well as managing the different co-existing versions of services and workflows. In this process, however, we focus on the discussion of the distributed deployment of the service line across multiple tiers.

During the deployment activity, the SaaS operator has to take several requirements into account. First, because of the multi-tenant context the services and components are shared among multiple tenants, and may perform tenant-

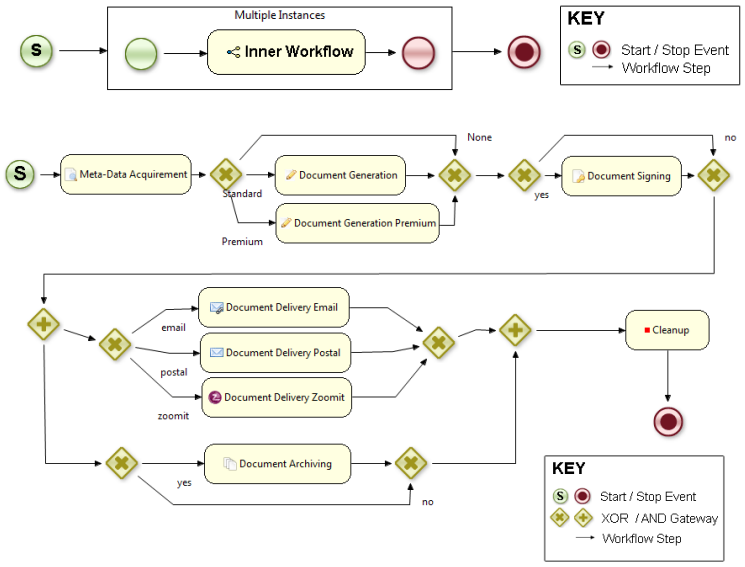

Figure 10: Process definition of the outer workflow (Fig. 10(a) at the top) and inner workflow (Fig. 10(b) at the bottom) for the document processing application [11].

specific behaviour. To enable tenant-aware customization, the different services and components (distributed across multiple tiers) should have *access to the tenant-specific software configurations*. One way to achieve this goal is hosting a centralized service that manages the configurations and provides an interface to the different components of the application to query for them. Alternatively, the software configurations can be stored in a scalable, replicated storage service that is accessible by all tiers of the application. Evidently, a library should be available on every node to ensure that the components can query specific configuration parameters.

Secondly, the different (sub)processes of the application may perform operations that impose *different resource needs*. For example, lightweight processes that are dispatching calls to other processes have different needs than processes that are involved in I/O access or heavy computation (e.g. for applying cryptography). The SaaS operator should therefore aim to combine a high utilization on the one hand and sufficient performance on the other hand – that is rather an optimization challenge where context-specific trade-offs depict the best achievable. One way to approach this optimization is to deploy and replicate processes independent from each other. Note, however, that sometimes some processes require hardware-specific support (e.g. for encryption) and cannot be deployed on any node (thus possibly only limited resources available).

Deployment design. The document processing service line consists of two types of processes to be deployed: the workflow engine and the back-end services. After a workflow instance is started for a tenant, the workflow engine

dispatches the control to the back-end services that are responsible for a certain activity (described in the workflow), and then (passively) waits for the work to be completed. After completion, control returns to the workflow engine, which then dispatches it to the next back-end service, and so on. During the execution of an entire workflow, the processing state as well as intermediate results are maintained by the workflow engine; the back-end services do not maintain state but (in contrast to the workflow engine) execute resource-intensive operations (e.g. generating or signing documents).

Fig. 11 presents the deployment diagram for the document processing service line. We allocated individual (virtual) nodes to the workflow engine and the back-end services. Furthermore, both types of processes are replicated independently of each other (i.e. independent multiplicity). This design is motivated by the different structural characteristics of both process types. The workflow engine is a rather lightweight (not CPU-intensive) process that requires low latency to enable the quick receiving and acknowledging of requests from end users (of tenants). Therefore, several instances of the workflow engine can be deployed in order to spread the load of incoming requests.

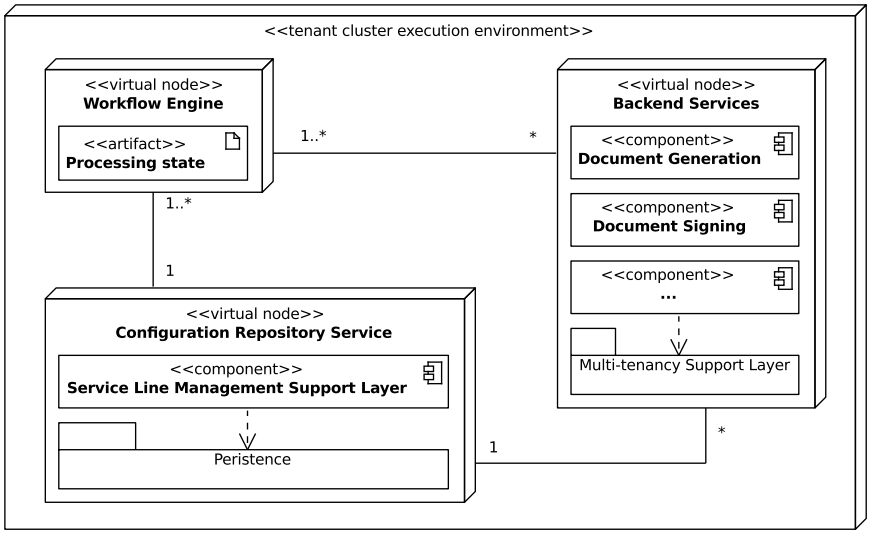

Figure 11: Deployment diagram for the Document Processing Service Line.

On the other hand, the back-end services are resource-intensive and should be optimized to achieve high throughput rather than low latency. Therefore, a sufficient amount of nodes should be available to deploy the back-end services, and possibly to scale up with increasing load. We decided to use uniform nodes that are able to execute any document processing service to support easy scalability. Although all document processing services are installed on those nodes, this does not necessarily mean that every service needs to be activated. Furthermore, these back-end services rely on a middleware layer to enable tenant-specific customization.

However, a coordination and resource management system is required to support independent multiplicity between the different types of processes, i.e. mon-

itoring the load and availability of the different nodes, scaling up/down with the load etc. We assume that the underpinning infrastructure (e.g. in the form of a PaaS offering) supports this, and we did not depict it in the deployment diagram.

The access to tenant-specific software configurations is provided by means of a centralized configuration repository service. The latter provides several interfaces to the workflow engine(s) and the back-end services to query for specific software configurations. We decided to use a single, centralized service in order to ensure strong consistency with respect to the storage of the tenant-specific configurations. To achieve consistency throughout the SaaS application, each request is tagged with extra meta information, specifying the unique tenant ID and the configuration version. This meta-data propagates with the message flow initiated by the request throughout the application and ensures that the appropriate tenant-specific configuration is activated at every tier. By using a version number, tenants can dynamically update their configuration without interfering ongoing requests. This coordination mechanism is based on the work in [23]. Evidently, when the amount of tenants increases, a replication strategy has to be worked out for the configuration repository (e.g. a master-slave setup).

The deployment design described above is used for each cluster of tenants of the document processing SaaS application. A separate deployment for each tenant cluster provides a lot of potential for efficient customization, tackling the trade-off between high reuse and high flexibility.

Deployment of the Prototype. The prototype has been deployed on top of a JBoss AS 7 cluster, and as workflow engine we used jBPM. The configuration respository is implemented by the *Service Line Management Support Layer*, as presented in Section 5, on top of a MySQL database. Customization of the workflow for document processing has been implemented using Drools, based on the approach described in [10]. To achieve application-level multi-tenancy and tenant-aware dynamic composition within services, we rely on the modular middleware layer that we presented in previous work ([24]). To realize the throughput SLAs, we have provided alternative service implementations that each guarantee a certain throughput (premium versus normal).

Since the prototype itself has a small scale, it was sufficient to allocate a single node for each of the different processes, i.e. the workflow engine, the configuration repository service, and the back-end services.

7 Provisioning New Tenants

This section discusses the different SLE activities for the scenario in which a new tenant wants to use the document processing service line developed in Section 6, and customize it to its requirements. We focus on the case that the service line already covers the requirements of the new tenant. Evidently, before a tenant administrator can start customizing the service line, he should register the tenant and create an administrator account. The tenant administrator can do this online, without intervention from the SaaS provider (cf. self-service).

7.1 Activity 4: Tenant Requirements Analysis

During the activity of tenant requirements analysis, tenant administrators use the `ITenantManagement` configuration interface (see Section 5) to select and parameterize features based on their requirements. The output of this activity is a tenant-specific feature configuration, containing the set of selected features, and optionally the associated attributes and their values (cf. Fig. 4).

In our prototype, we created a feature configuration for Tenant A and for Tenant B, by selecting features based on their requirements as described in Section 6.1. Fig. 12 shows a screenshot of the configuration interface in our prototype while selecting the features for Tenant B. This activity resulted in the following two feature configurations[4]:

```
Tenant A:
   REST
   Flag_per_Document
   PDF
   Separate_File
   Premium
   Paper

Tenant B:
   SFTP
   Flag_per_Document
   XML
   Extract_from_Data
   Invoice_Template
   Signing cert: TenantB-doc.cert
   Standard
   Email
   Paper
   Archival duration: 24
```

7.2 Activities 5 & 6: Configuration Mapping and Activation

After the feature configurations are specified and confirmed by the tenant administrator, the two remaining activities in the SLE method are executed automatically and do not require any human intervention.

The feature configurations are *automatically* transformed into tenant-specific software configurations by applying the feature-to-software-composition mappings that have been defined during the design activity (see Section 6.2). The end result, a software configuration (cf. Fig. 6), binds a specific variant to at least each mandatory variation point in the service line. After this configuration mapping activity is finished, the tenant-specific software configurations are available in the service line and end users of Tenants A and B can thus immediately start using the document processing application.

The actual activation of a particular software configuration and the necessary variants occurs at run time, based on the incoming request. For example,

[4]The version numbers of the features are not presented, as all features have version 1.

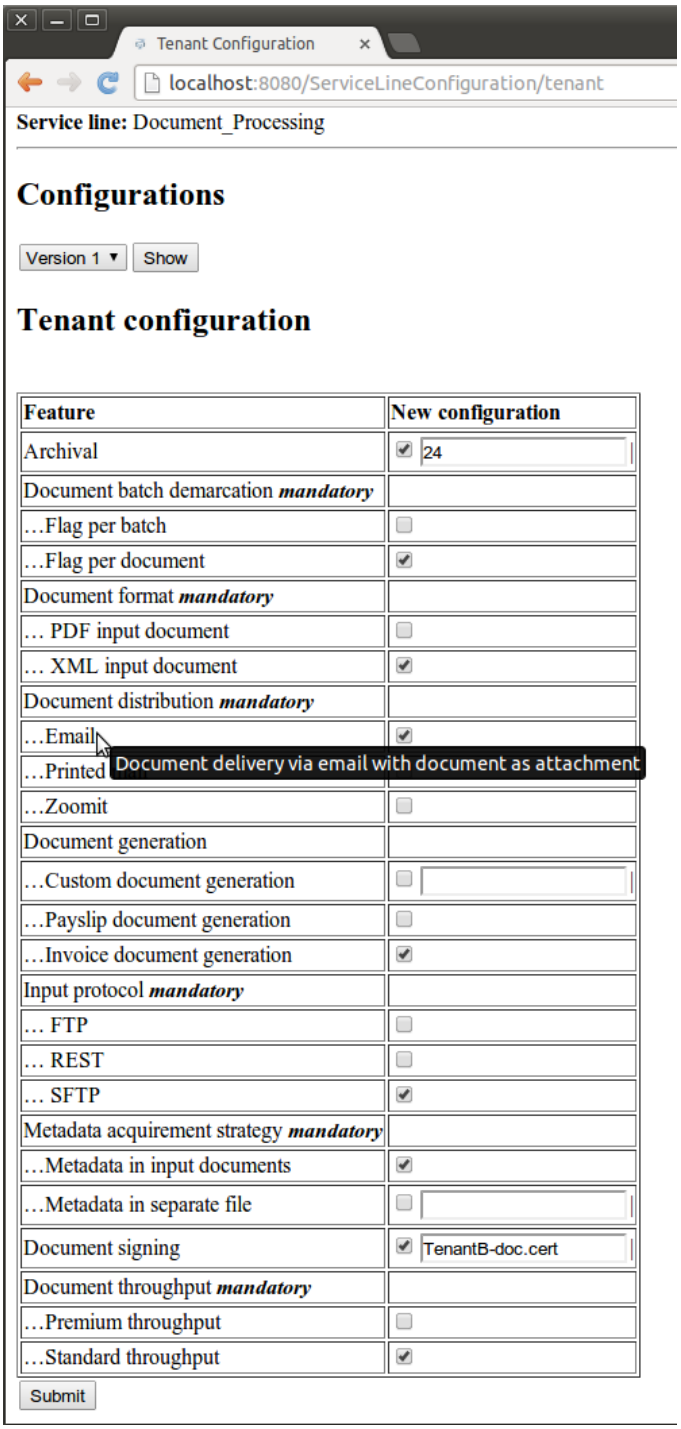

Figure 12: Screenshot of the configuration interface.

when Tenant A submits a batch of documents, the appropriate software configuration is fetched and activated. Next, the `Preprocessing` service activates the right variants when processing these input documents. Further, the service line activates the appropriate services in the `DocumentProcessing` workflow. For example, documents will be distributed using the `PostalService` variant for the `DeliveryChannel` variation point (corresponding to the `Paper` feature). The document is first sent to the printing service and next the printed document is delivered via the postal service to the appropriate recipient's address. In comparison, the document processing workflow for Tenant B also consists of services for document generation, signing and archival.

In the meantime, other tenants with different requirements can simultaneously use the document processing service line. These different tenant-specific configurations co-exist in the service line, and thus these tenants and their end users are serviced simultaneously.

8 Update Tenant-specific Requirements

Evidently, the requirements of a tenant can change over time. To satisfy these varying requirements, the tenant administrator has to update the tenant-specific feature configuration. Based on this updated feature configuration and the existing feature-to-software-composition mappings, a new tenant-specific software configuration is generated. This new software configuration is immediately effective, without any manual effort or human intervention. Thus, updating requirements is comparable to provisioning a new tenant (cf. Section 7) and includes the activities of tenant requirements analysis, configuration mapping, and configuration activation.

For example, Tenant A now also wants to archive the payslips that have been processed. The tenant administrator of Tenant A selects the `Archival` feature and fills in the duration of the archival period (e.g. 12 months). The `ITenantManagement` configuration interface will report back to the tenant administrator that, in order to select `Archival`, the `Signing` feature is also required. The tenant administrator then also selects `Signing`, and (in contrast to Tenant B) decides to use the default certificate of UnifiedPost instead of a custom one. The updated feature configuration is as follows:

```
Tenant A:
   REST
   Flag_per_Document
   PDF
   Separate_File
   Signing cert: TenantB-doc.cert
   Premium
   Paper
   Archival duration: 12
```

9 Supporting New Requirements

Some requirements of existing or new tenants are not yet covered by the current version of the service line. If these new requirements can be satisfied, this implies

that support for these new requirements should be introduced into the service line. Evidently, these changes can have an impact on the existing workflows, services and components, as well as the different configurations and mappings.

Tenant B has new requirements that cannot be addressed by the current document processing application. Therefore, the tenant administrator of Tenant B sends a feature request, containing a description of the new requirements, to the SaaS provider. This feature request results in a re-iteration of the development process, starting with the domain analysis.

9.1 Activity 1: Domain Analysis

The internal business process of Tenant B has been adapted and the format of raw invoice data that is used as input for document processing, has been changed to CSV. Furthermore, Tenant B wants to be sure that every invoice has been delivered. Therefore, in the case of document delivery via email, a link to the document is provided (instead of an attachment), which enables the customer, after authenticating, to view and download the document. However, if after 48 hours the document has not been retrieved, the particular document should still be printed and sent via printed mail.

With respect to our prototype, we extended the feature model to address the new requirements (and possible other extensions based on these new requirements). Fig. 13 depicts the extended feature diagram. We added `CSV` as a new sub-feature for the format of input data, and created a `Cascaded_Delivery` delivery channel. In addition to Tenant B's requirement enable to switch to printed mail in case of a timeout, we also included the option to switch to printed mail in case of a failure while sending an email (e.g. non-existing recipient's address). `Email_and_Paper_after_Deadline` also has an attribute to define the timeout period. Finally, the *Email* feature has been split up in two sub-features: emails can be sent with the document in attachment, or with a link to the document in UnifiedPost's document store.

9.2 Activity 2: Service Line Architecture Design & Implementation

Supporting the CSV format in the service line architecture only requires an additional variant in the `Preprocessing` service for input data format. Thanks to the modular implementation of the software variants, this component can be easily added without impact on the other software variants.

Cascaded delivery, however, is designed as a separate workflow (i.e. service orchestration) that reuses the existing delivery channel services (i.e. email, postal and Zoomit service). This allows the service line to be further extended with other combinations of the existing delivery channels. Deciding when another delivery channel needs to be used is decided by a decision component in which different algorithms can be plugged in (e.g. timeout- and failure-driven). Although this feature is implemented as a workflow, it is in its entirety a self-contained service that can be plugged into the document processing workflow.

To support the two types of sending email, a new version of the email service is implemented. This new version has a variation point for document provisioning and two variants (i.e. provisioning via attachment or link).

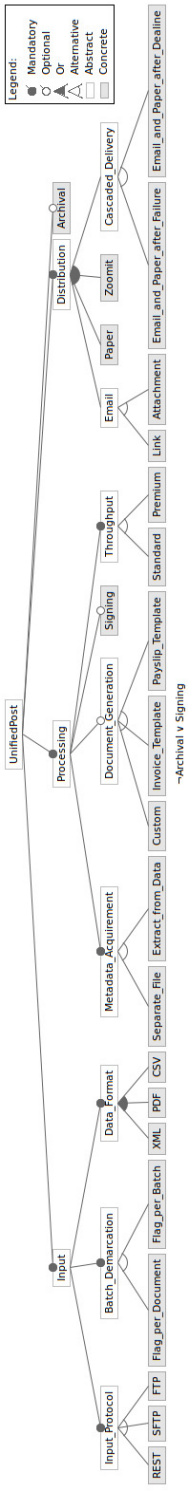

Figure 13: Extended feature diagram of the document processing SaaS application.

Furthermore, additional feature mappings have been specified to support the new features (also see Appendix A): Listings 12, and 13 respectively contain the mapping for the `CSV` and the `Cascaded_Delivery` features, while Listing 3 *replaces* the mapping for the original `Email` feature (in Listing 11). Notice that the mappings of the `Cascaded_Delivery` features refer to version 2 of the email service (see lines 6 and 23).

Listing 3: Feature mappings for the updated Email sub-features.

```
1  featuremapping EmailWithLink {
2      feature Link-v1;
3      composition DocProcessingWorkflow {
4          binding {
5              variationPoint: DeliveryChannel;
6              variant: EmailService-v2;
7          }
8          composition Email;
9      }
10     composition Email {
11         binding {
12             variationPoint: DocumentProvisioning;
13             variant: DocumentViaLink-v1;
14         }
15     }
16 }
17
18 featuremapping EmailWithAttachment{
19     feature Attachment-v1;
20     composition DocProcessingWorkflow {
21         binding {
22             variationPoint: DeliveryChannel;
23             variant: EmailService-v2;
24         }
25         composition Email;
26     }
27     composition Email {
28         binding {
29             variationPoint: DocumentProvisioning;
30             variant: DocumentAttached-v1;
31         }
32     }
33 }
```

9.3 Activity 3: Service Line Deployment & Operation

Before updating the running service line, the SaaS operator should verify that the new updates do not affect the current configurations of the different tenants (via the traceability support). The only new requirement that did have an impact on the existing features and/or their implementation is the requirement to send emails with a link to the document. After the update, there exists no mapping any more to the *Email* feature, only to its sub-features. Because the original feature is actually replaced by the `Attachment` sub-feature, this means that all feature configurations that rely on this feature need to be adapted. The SaaS operator can quickly verify to which tenants this is applicable (in this example Tenant B) and update the feature configurations. Automatically, the

new software configuration is then generated.

As the other new requirements in our example did not have an impact on the mappings or implementation of the other features, the update can now be executed. In our prototype we did not provide support for rolling upgrades [7], nor dynamic updates. Thus, we performed the complete update offline. This example, however, illustrates the need for a gradual roll-out of upgrades for high available SaaS applications: on the one hand the SaaS operator cannot perform the update on the service line when some tenants still rely on the older version; on the other hand, updating the feature configuration before updating the service line also results in run-time conflicts. A gradual roll-out allows to temporarily run the two versions of the email service next to each other, which gives the SaaS operator the opportunity to update the feature configurations.

The option to deploy multiple versions of the service line is sometimes necessary when an update introduces a conflict that cannot be easily solved. The affected tenants can then be warned in order to upgrade them to the next version of the service line. In the meantime, the other tenants can already use the newest version of the SaaS application.

This example clearly shows that the constraints that are imposed by our SLE method show their benefits when extending the service line. Composeability enables the SaaS provider to easily support new requirements, while the traceability concern (i.e. versioning) allows to locate issues quickly and to support multiple co-existing versions of the service line.

9.4 Activity 4: Tenant Requirements Analysis

After the service line is updated, the different tenant administrators can select the new features via the `ITenantManagement` configuration interface. The tenant administrator of Tenant B can now adapt his feature configuration to satisfy the new requirements of Tenant B. This results in the following feature configuration:

```
Tenant B:
   SFTP
   Flag_per_Document
   CSV
   Extract_from_Data
   Invoice_Template
   Signing cert: TenantB-doc.cert
   Standard
   Email_and_Paper_after_Deadline timeout: 48
   Paper
   Archival duration: 24
```

9.5 Activities 5 & 6: Configuration Mapping and Activation

When a user of Tenant B now submits new input data (in CSV format), the service line retrieves the most recent software configuration of Tenant B (i.e. highest version number). As a consequence, the document processing workflow does not directly call the email service any more (cf. the previous configuration), but uses

the cascaded delivery workflow. The latter will then call the email service, and after the timeout it will also call the postal service. Concretely, first emails are sent out containing a link to the generated invoices, and if a document is not retrieved after 48 hours, then that particular document is printed and sent via the postal service to the appropriate recipient.

10 Updating and Maintaining SaaS Applications

The SaaS operator is responsible for the maintenance of the SaaS application and for keeping it up-to-date. For example, the underpinning platform or used libraries have to be upgraded, bugs have to be fixed, etc. These changes can be encapsulated into a single software component, but at the other end of the spectrum, they can also affect the complete architecture of the SaaS application and even conflict with the requirements of some tenants. Furthermore, as it is a key advantage of SaaS applications that updates are performed by the SaaS provider (and not by the tenants individually), SaaS applications are continuously updated to the newest version.

Similarly to Section 9, the SaaS operator has to verify the different configurations and correct them where necessary. Again there is the option to (temporarily) deploy multiple versions in case of conflicts.

We assume that the most common updates are related to bug fixes in and updates to the implementation of the software variants (e.g. performance improvements or new libraries). In this case, updates are localized to single features or software variants, and thus have minimal impact. For example, a new version of the library for generating payslips is released. The third mapping in Listing 7 should then be updated to the `UP_PayslipGeneration-v2` variant. Consequently, new software configurations are automatically generated for all feature configurations that contain the `Payslip` feature.

11 Conclusion

This report demonstrates how the SLE method was applied to develop and manage a customizable, multi-tenant SaaS application in an efficient way. We developed a prototype in collaboration with an industrial partner in the domain of online document processing, showing the feasibility and practical realizability of building an integrated service line. First, we presented some application artifacts that illustrate how a service line is developed and deployed, as well as how tenants can configure it tailored to their specific requirements and preferences. Further, we illustrated how evolution and maintenance of a service line can be addressed using our method. Finally, we proposed several auxiliary solutions to complement the generic SLE method in order to make it effective in practice. These auxiliary elements are reusable and showcase how the imposed constraints can be addressed.

References

[1] Frederick Chong and Gianpaolo Carraro. Architecture Strategies for Catching the Long Tail. Microsoft Corporation, `http://msdn.microsoft.com/en-us/library/aa479069.aspx`, April 2006.

[2] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76(12):1130–1143, 2011. Special Issue on Software Evolution, Adaptability and Variability.

[3] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[4] CUSTOMSS. CUSTOMization of Software Services in the cloud (iMinds ICON project). `http://www.iminds.be/en/research/overview-projects/p/detail/customss`, 2011. [Last visited on November 20, 2013].

[5] Krzysztof Czarnecki, Michal Antkiewicz, and Chang Hwan Peter Kim. Multi-level customization in application engineering. *Commun. ACM*, 49(12):60–65, Dec 2006.

[6] Linda DeMichiel. JSR 317: Java^{TM} Persistence 2.0. `http://www.jcp.org/en/jsr/detail?id=317`, 2009. [Last visited on 20 November 2013].

[7] Tudor Dumitraş and Priya Narasimhan. Why do upgrades fail and what can we do about it? In JeanM. Bacon and BrianF. Cooper, editors, *Middleware '09: 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 349–372. Springer Berlin Heidelberg, 2009.

[8] A. Etien and C. Salinesi. Managing requirements in a co-evolution context. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering* , pages 125–134. IEEE, 2005.

[9] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. `http://martinfowler.com/articles/injection.html`, January 2004.

[10] Kristof Geebelen, Stefan Walraven, Eddy Truyen, Sam Michiels, Hendrik Moens, Filip De Turck, Bart Dhoedt, and Wouter Joosen. An open middleware for proactive QoS-aware service composition in a multi-tenant SaaS environment. In *ICOMP '12: Proceedings of the 2012 International Conference on Internet Computing*. CSREA Press, July 2012.

[11] Fatih Gey, Stefan Walraven, Dimitri Landuyt, and Wouter Joosen. Building a customizable Business-Process-as-a-Service application with current state-of-practice. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *SC '13: 12th International Conference on Software Composition*, pages 113–127. Springer Berlin / Heidelberg, 2013. doi: 10.1007/978-3-642-39614-4_8.

[12] Google, Inc. Google App Engine. `http://code.google.com/appengine/`. [Last visited on 20 November 2013].

[13] Chang Jie Guo, Wei Sun, Ying Huang, Zhi Hu Wang, and Bo Gao. A framework for native multi-tenancy application development and management. In *CEC/EEE '07: 9th IEEE International Conference on E-Commerce Technology and 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, pages 551–558, July 2007. doi: 10.1109/CEC-EEE.2007.4.

[14] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report 21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[15] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, 2002. doi: 10.1109/MS.2002.1020288.

[16] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A tool framework for feature-oriented software development. In *ICSE '09: 31st IEEE International Conference on Software Engineering*, pages 611–614, May 2009. doi: 10.1109/ICSE.2009.5070568.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP '97 - Object-Oriented Programming*, pages 220–242. Springer Berlin Heidelberg, 1997. doi: 10.1007/BFb0053381.

[18] P. B. Kruchten. The 4+1 View Model of architecture. *IEEE Software*, 12(6):42–50, 1995. doi: 10.1109/52.469759.

[19] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–119, 2001. doi: 10.1109/2.910904.

[20] Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. In *PLDI '11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 425–436, New York, NY, USA, 2011. ACM. doi: 10.1145/1993498.1993548.

[21] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer-Verlag New York Inc, 2005.

[22] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, Boston, MA, USA, Second edition, 2002.

[23] Stefan Walraven, Bert Lagaisse, Eddy Truyen, and Wouter Joosen. Policy-driven customization of cross-organizational features in distributed service systems. *Software: Practice & Experience*, 43(10):1145–1163, October 2013. doi: 10.1002/spe.1128.

[24] Stefan Walraven, Eddy Truyen, and Wouter Joosen. A middleware layer for flexible and cost-efficient multi-tenant applications. In *Middleware '11: Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, pages 370–389. Springer Berlin / Heidelberg, 2011. doi: 10.1007/978-3-642-25821-3_19.

[25] Stefan Walraven, Eddy Truyen, and Wouter Joosen. Comparing PaaS offerings in light of SaaS development. *Computing*, pages 1–56, 2013. doi: 10.1007/s00607-013-0346-9.

[26] Stefan Walraven, Dimitri Van Landuyt, Eddy Truyen, Koen Handekyn, and Wouter Joosen. Efficient customization of multi-tenant Software-as-a-Service applications with service lines. *J. Syst. Software*, 2014. doi: 10.1016/j.jss.2014.01.021.

A Feature-to-software-composition Mapping Specifications in the Prototype

This section shows all remaining feature-to-software-composition mappings that have been specified for our prototype and that have not been included in the report for the sake of readibility.

Listing 4: Feature mappings for the Input Protocol sub-features.

```
1  featuremapping Input_REST {
2      feature REST-v1;
3      composition Preprocessing {
4          binding {
5              variationPoint: InputProtocol;
6              variant: REST-v1;
7          }
8      }
9  }
10
11 featuremapping Input_SFTP {
12     feature SFTP-v1;
13     composition Preprocessing {
14         binding {
15             variationPoint: InputProtocol;
16             variant: SFTP-v1;
17         }
18     }
19 }
20
21 featuremapping Input_FTP {
22     feature FTP-v1;
23     composition Preprocessing {
24         binding {
25             variationPoint: InputProtocol;
26             variant: FTP-v1;
27         }
28     }
29 }
```

Listing 5: Feature mappings for the Batch Demarcation sub-features.

```
featuremapping Demarcation_FlagPerDoc {
    feature Flag_per_Document-v1;
    composition Preprocessing {
        binding {
            variationPoint: DemarcationAlgorithm;
            variant: FlagPerDoc-v1;
        }
    }
}

featuremapping Demarcation_FlagPerBatch {
    feature Flag_per_Batch-v1;
    composition Preprocessing {
        binding {
            variationPoint: DemarcationAlgorithm;
            variant: FlagPerBatch-v1;
        }
    }
}
```

Listing 6: Feature mappings for the Data Format sub-features.

```
featuremapping Format_XML {
    feature XML-v1;
    composition Preprocessing {
        binding {
            variationPoint: InputFormat;
            variant: XMLParser-v1;
        }
    }
}

featuremapping Format_PDF {
    feature PDF-v1;
    composition Preprocessing {
        binding {
            variationPoint: InputFormat;
            variant: PDFReader-v1;
        }
    }
}
```

Listing 7: Feature mappings for the Document Generation sub-features.

```
featuremapping DocumentGeneration_Custom {
    feature Custom-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: DocumentGeneration;
            variant: CustomDocGenerationService-v1;
            attribute: template;
        }
    }
}

featuremapping DocumentGeneration_Invoice {
    feature Invoice-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: DocumentGeneration;
            variant: UPDocGenerationService-v1;
        }
        composition DocGeneration;
    }
    composition DocGeneration {
        binding {
            variationPoint: DocumentTemplate;
            variant: UP_InvoiceGeneration-v1;
        }
    }
}

featuremapping DocumentGeneration_Payslip {
    feature Payslip-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: DocumentGeneration;
            variant: UPDocGenerationService-v1;
        }
        composition DocGeneration;
    }
    composition DocGeneration {
        binding {
            variationPoint: DocumentTemplate;
            variant: UP_PayslipGeneration-v1;
        }
    }
}
```

Listing 8: Feature mapping for the Signing feature.

```
featuremapping DocumentSigning {
    feature Signing-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: Signing;
            variant: DocumentSigningService-v1;
            attribute: cert;
        }
    }
}
```

Listing 9: Feature mappings for the Throughput sub-features.

```
featuremapping Throughput_Standard {
    feature Standard-v1;
    composition DocumentProcessing {
        binding {
            variationPoint: DocProcessingType;
            variant: StandardDocProcessingWorkflow-v1;
        }
        composition DocProcessingWorkflow;
    }
}

featuremapping Throughput_Premium {
    feature Premium-v1;
    composition DocumentProcessing {
        binding {
            variationPoint: DocProcessingType;
            variant: PremiumDocProcessingWorkflow-v1;
        }
        composition DocProcessingWorkflow;
    }
}
```

Listing 10: Feature mapping for the Archival feature.

```
featuremapping Archive {
    feature Archival-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: Archival;
            variant: DocumentArchivalService-v1;
            attribute: duration;
        }
    }
}
```

Listing 11: Feature mappings for the Distribution sub-features.

```
featuremapping EmailWithAttachment {
    feature Email-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: DeliveryChannel;
            variant: EmailService-v1;
        }
    }
}

featuremapping PrintedMail{
    feature Paper-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: DeliveryChannel;
            variant: PostalService-v1;
        }
    }
}

featuremapping Zoomit{
    feature Zoomit-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: DeliveryChannel;
            variant: ZoomitService-v1;
        }
    }
}
```

Listing 12: Feature mapping for the CSV Data Format sub-feature.

```
featuremapping Format_CSV {
    feature CSV-v1;
    composition Preprocessing {
        binding {
            variationPoint: InputFormat;
            variant: CSVParser-v1;
        }
    }
}
```

Listing 13: Feature mappings for the Cascaded Delivery sub-features.

```
featuremapping CascadedDelivery-Email-Paper-Failure {
    feature Email_and_Paper_after_Failure-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: DeliveryChannel;
            variant: CascadedDeliveryWorkflow-v1;
        }
        composition: CascadedDelivery;
    }
    composition CascadedDelivery {
        binding {
            variationPoint: FirstChannel;
            variant: EmailService-v2;
        }
        binding {
            variationPoint: SecondChannel;
            variant: PostalService-v1;
        }
        binding {
            variationPoint: DecisionAlgorithm;
            variant: FailureBasedDecisionAlgorithm-v1;
        }
        composition Email;
    }
    composition Email {
        binding {
            variationPoint: DocumentProvisioning;
            variant: DocumentViaLink-v1;
        }
    }
}

featuremapping CascadedDelivery-Email-Paper-Deadline {
    feature Email_and_Paper_after_Deadline-v1;
    composition DocProcessingWorkflow {
        binding {
            variationPoint: DeliveryChannel;
            variant: CascadedDeliveryWorkflow-v1;
        }
        composition: CascadedDelivery;
    }
    composition CascadedDelivery {
        binding {
            variationPoint: FirstChannel;
            variant: EmailService-v2;
        }
        binding {
            variationPoint: SecondChannel;
            variant: PostalService-v1;
        }
        binding {
            variationPoint: DecisionAlgorithm;
            variant: TimeoutBasedDecisionAlgorithm-v1;
            attribute: timeout;
        }
        composition Email;
    }
    composition Email {
        binding {
            variationPoint: DocumentProvisioning;
            variant: DocumentViaLink-v1;
        }
    }
}
```